TACTICS for User Interface Design: Coupling the Compositional and Transformational Approach

Srdjan Kovacevic
U S WEST Advanced Technologies
4001 Discovery Drive, Boulder, CO 80303
srdjan@advtech.uswest.com

ABSTRACT

The paper describes TACTICS, the model and a model-based tool capable of supporting a wide range of design decisions and providing assistance in the design process. The TACTICS tool automatically generates a user interface for an application and assists in refining it and in detecting and resolving design inconsistencies. The TACTICS model of human-computer interaction integrates a compositional model of UIs and a transformational model of the UI design space. A user interface is viewed as a *composition* of primitives structured based on the application and on the desired dialogue style, and the model identifies user interface components and structuring principles for assembling components into a coherent interface. The model also defines *transformations* that modify UI structures to achieve a desired look-and-feel and enable designers to easily explore different UI designs. The paper describes the knowledge structure of the model and the TACTICS approach to generating user interfaces. UI components are discussed and examples of UI structures given.

Keywords: UI model, model-based design, UI design tools, UI components, UI representation, automatic generation, design transformations.

1. INTRODUCTION

The capabilities of a UI design tool are limited by its underlying model. Features of a UI that are not recognized and explicitly represented by the model are not manipulable by the tool either. Each such feature results in a "hardwired" solution in all UIs produced by the tool, thus leaving it beyond direct designer's control and limiting the range of design decisions the tool supports. Navigational support is affected as well – without knowledge about specific features of a UI (e.g. command syntax), and possible variations (e.g. prefix, postfix), the tool cannot assist in changing designs with respect to this feature.

Navigational support includes not only assistance in moving from one point in the design space to another, but guidance in the process. It is especially important when designing complex UIs, where features interact and changing one aspect of a UI affects others. Again, unless the model captures all features and their interdependencies, the tool cannot offer this level of navigational support.

Early UI research was based on the premise that an application's UI can be isolated from the application's functionality, allowing for development of different UIs for an application without affecting its non-interactive part. The Seeheim model [Green 85] is representative of this traditional approach. However, separation inherently limits the range of interfaces that can be produced; in particular, interfaces providing semantic feedback are not possible without access to the application semantics. Accordingly, the underlying model must capture enough application semantics for a class of UIs to be produced by a design tool.

This paper describes a new model of human-computer interaction which meets the above requirements. The TACTICS model supports (1) automatic generation of UIs, (2) a wide range of different designs, (3) easy transitions from one design to another, and (4) integration of different kinds of knowledge needed for guiding a UI designer. TACTICS, which stands for ***T**ransformation- **A**nd **C**omposition-based **T**ool for **I**nterface **C**reation and **S**upport*, integrates a compositional model of user interfaces and a transformational model of the UI design space. The model provides a synthesis of application semantics and the UI domain knowledge. The TACTICS model is *compositional* as it views a user interface as a collection of primitives structured based on the application and on the desired dialogue style; the model identifies user interface components and structuring principles for assembling components into a coherent interface. The behavior and the look-and-feel of a UI are explained in terms of properties of UI components and the way they are structured. The model is *transformational* as it views the UI design space as a grid and defines a set of transformations for moving along the grid lines that connect different UI designs. Transformations modify UI structures to achieve a different look-and-feel. They are a vehicle for exploring the design space, making it easy for a designer to generate and try alternative designs. The model also integrates additional knowledge for providing assistance in this process.

Automatic generation in most of the UI tools comes at the

expense of a limited flexibility and range of designs that can be produced. For instance, ITS [Wiecha 90], DON [Kim 90], and DeBaar's integration of D2M2 and DevGuide [DeBaar 92] focus on the creation of dialogue boxes for graphical UIs, but do not address other components needed for different dialogue styles. Mickey [Olsen 89] automatically generates menu and dialogue box-based UIs, Cousin [Hayes 85] and Mecano [Puerta 92] generate only form-based UIs, while Diction [Singh 89] supports menu-based UIs. By coupling the compositional and the transformational approach to UI design, TACTICS can provide the automatic generation and yet support a wide range of designs and easy transitions from one design to another.

With a few exceptions, other UI tools do not provide a specific support for changing designs. Diction allows a designer to change command syntax easily, as it captures a notion of command syntax and the mapping between different syntaxes. ITS allows a designer to change style rules that establish global policies concerning generation of dialogue boxes and menus. Mecano includes graphical editing capabilities to modify generated forms and dialogue boxes; it also keeps track of edits and can reapply them to later revisions of the same UI. UIDE [Foley 91] goes furtherst and provides a set of built-in transformations specifically aimed to make switching from one design to another easy. TACTICS builds on UIDE, by extending its model and its set of transformations.

The compositional approach is not new either. Szekely [Szekely 91] models interactive programs based on the notion of communication concepts representing the information that users and programs can communicate, but he focuses only on a class of graphical user interfaces. Humanoid [Szekely 92] applies the compositional approach to creating presentations based on a template-based model. Others have modeled other UI components [Card 90, Bleser 90, Myers 90]. However, they do not model the overall UI as an integral structure.

The most advanced model-based UI tools to date are the User Interface Design Environment (UIDE) [Foley 91, 91a] and Humanoid [Szekely 92]. Cartoonist [Sukaviriya 90], which is developed based on the UIDE framework, also belongs here. UIDE does not have a full-fledged UI representation model, but the application model is extended with some UI-specific details and used as an executable specification.

Cartoonist expands the original UIDE model with UI concepts such as interaction techniques, interface actions, and interface objects, but falls short of developing a full-fledged UI representation model. Both Cartoonist and Humanoid utilize the same approach as UIDE and use the application description, extended with UI-specific details, as an executable specification. In addition, Humanoid lacks the explicit data model and its control model is not completely declarative. Consequently, Humanoid is not well suited for providing design assistance that requires reasoning about the UI and application properties. MASTERMIND [Neches 93] is an ongoing effort to integrate the strengths of UIDE and Humanoid.

TACTICS goes beyond previous tools and models because (1) it identifies a more complete set of UI components, not limited to interaction objects and techniques, and (2) it couples the compositional and transformational approach. New components, especially buffering components, increase reusability and flexibility of a UI, while also increasing power of transformations.

The paper is organized as follows. Section 2 gives an overview of the TACTICS model. Section 3 discusses the TACTICS approach to generating and exploring UIs. An example illustrating the concepts behind this approach is introduced in Section 4. Section 5 focuses on the UI model. Section 6 discusses UI components and expands on the example from Section 4. Implementation is briefly discussed in section 7, followed by conclusions.

2. TACTICS MODEL

The TACTICS model integrates several different kinds of knowledge, as shown in Figure 1. Size of regions does not necessarily correlate to the importance or the volume of the knowledge they represent, but their intersections symbolize interactions between different kinds and forms of knowledge.

To generate and manage a UI for an application, a tool must know about the application – what are the application's communication needs that a UI must meet. Application semantics are captured in the TACTICS model using modeling primitives based on the UIDE model [Foley 91]. An application is described in terms of its objects and their properties, actions that can be applied to these objects, information required by each action (action parameters), and action pre- and post-conditions. TACTICS extends the UIDE model with additional relationships linking different objects and their properties, and relationships between action parameters [Kovacevic 92a].

The knowledge about the application semantics is necessary, but not sufficient for creating an application UI. Also required is adequate user interface domain knowledge – the knowledge about UI components and valid UI structures. The general UI domain knowledge is captured in two

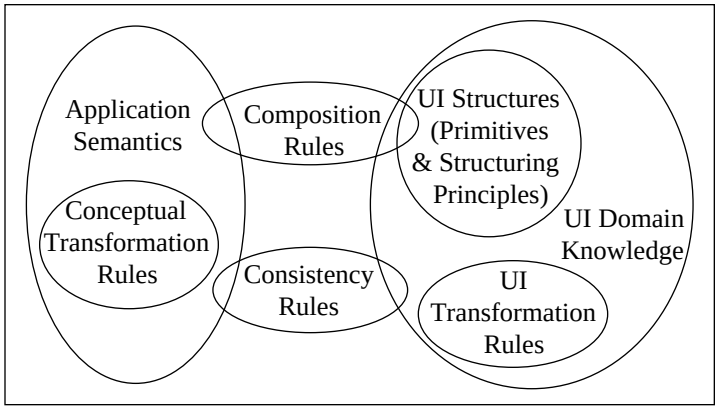

Figure 1. Knowledge structure of the TACTICS model

forms: one is a set of UI primitives and basic structuring principles, and the other is the set of rules for transforming UI structures. The UI components and structuring principles for assembling these components are discussed in the following sections.

Application semantics can be represented independent from the UI domain knowledge; hence, they are represented by disjoint regions in Figure 1. However, for the automatic generation of UIs, the two must be integrated. The TACTICS model provides such a synthesis of the application semantics and the UI domain knowledge. A relation between the application semantics and the corresponding UI is captured in two sets of rules: a set of composition rules, and a set of consistency rules for conflict detection and resolution. The composition rules explicate mappings from the structure representing the application semantics to the structure representing the application's UI; they establish correspondence between the application modeling and UI modeling primitives. The consistency rules evaluate a UI structure corresponding to an application with respect to the application's semantics to check the UI validity according to some criteria; they detect conflicts and assist in their resolution.

Generally, there may be more than one user interface structure satisfying communication needs of an application. TACTICS captures relationships between these UIs in transformation rules. Design transformations that modify the application conceptual model while preserving the application functionality were first introduced by UIDE [Foley 87]. Preserving the application functionality is important because it allows reuse of the application functional part. This means that no new procedural code has to be provided, and a UI designer can explore different designs without programmers' assistance. It also guarantees that an application has all required functionality as originally designed, regardless of its interface.

UIDE does not have a separate UI representation, and all of its transformations operate directly on the application conceptual model. TACTICS takes advantage of separate representations for application semantics and for the UI and introduces transformations that modify a UI while leaving the application conceptual model unchanged. Transformations that modify the conceptual model while preserving the application functionality are also kept, such as transformations for specializing application actions. The conceptual and UI transformations are represented as separate regions because they operate on different structures—the application conceptual model versus the UI structure—and do not establish a direct link between the two structures. This link is maintained by other rules.

3. GENERATING AND EXPLORING USER INTERFACES

The transformations that operate on UI structures complement the composition rules. The application conceptual model does not uniquely define the application's UI, since it does not specify UI details such as command syntax and interaction techniques to be used. Therefore, it allows a range of different UIs which, in turn, may each have a different look-and-feel. Unspecified UI details are considered *specification freedoms*. An application designer does not have to make all UI-related decisions up-front. Instead, the composition rules provide defaults and generate a default UI. The default UI is one of possibly many UIs that meet application requirements; all other designs are produced by applying transformations.

While transformations in software engineering have been used for generating "functional" code for a target program from a high-level specification and for performing optimizations [Balzer 85, Darlington 81, Partsch 83], UIDE was the first to apply the transformation approach to UI design. TACTICS improves on UIDE in two aspects. The first improvement is a separate UI representation capturing UI specifics at finer level of details than in UIDE. While UIDE keeps UI details together with the application conceptual model, which is then interpreted at run time, TACTICS generates a separate object-oriented structure implementing the application UI. Different UIs are produced by configuring individual components and relationships among them. This allows for finer control over UI features then is possible in UIDE. The second improvement is a wider range of transformations. TACTICS supports both transformations for fine-grain control over UIs, made possible by the finer UI representation, and high-level transformations for making more complex changes to UI design, such as changing one dialogue style to another. High-level transformations are themselves composed of lower-level transformations.

Specification freedoms are important because they free designers from having to make commitments early in the design process. As such, they have been used in other systems as well. For instance, HUMANOID [Szekely 92] also provides defaults for unspecified UI details, allowing designers to execute their design even before it is completely concretized. Designers can experiment with the system and incrementally refine it. These refinements are comparable to the low-level transformations in TACTICS. However, HUMANOID lacks the explicit support for the more complex changes, equivalent to higher-level transformations in TACTICS.

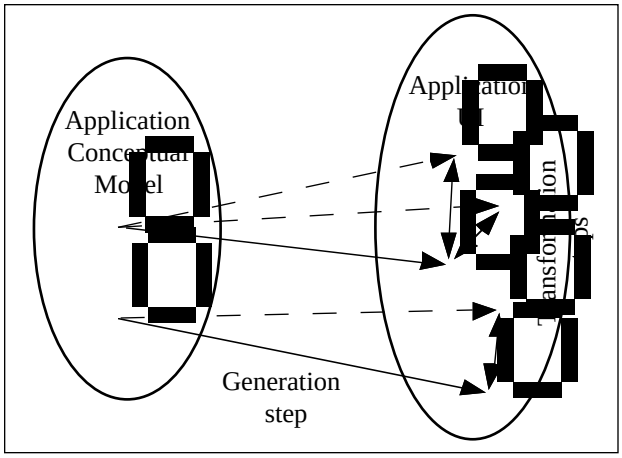

Figure 2. Generating UIs in TACTICS.

Figure 2 illustrates the TACTICS approach to generating and exploring an application's UI. The mapping from the application to its corresponding UI is decomposed into a generation step and transformation steps. The first step generates a default UI. The generation step is represented by solid arrows that connect points representing different applications with their default UIs. Other UIs, not connected directly with their corresponding application (connections represented by dashed arrows), are produced by modifying the default UI. These modifications map one UI into another one corresponding to the same application; they correspond to transformation steps and are represented by bidirectional arrows in Fig. 2.

The generation step is performed by composition rules, and transformation steps by transformation rules. It is important to note that no generality is lost by decomposing the mapping from an application to its UI into the generation and transformation steps. All possible UI designs can still be reached, as long as they can be built of UI components identified in the TACTICS model. The complexity of the system has not increased either; transformations would be needed anyway to support navigation in the UI design space and transitions from one design to another. Quite the contrary, the complexity of composition rules has been reduced, since they do not have to generate all possible UIs, but only a subset of them.

4. EXAMPLE

To illustrate the basic concepts of the TACTICS model, let us consider an example about designing a UI for a Circuit Design application. The circuit design program is a specialized drawing program where graphical objects have specific meanings: they represent logical components of electronic circuits, such as *NOT gate* and *NAND gate*, and lines represent connections between components. Components have layout related properties, such as *position* and *orientation*, but also properties with domain specific meanings, such as *fan-in* and *fan-out*. This example focuses on the *move-gate* action. It takes two parameters: a *gate* to be moved, and a new *position* of the gate.

Figure 3 illustrates this description of the action; at this level there are no details concerning the application UI. To generate a UI for the action, a tool complements this description with the UI domain knowledge, which defines requirements that such a UI must meet. Namely, a UI must enable users to select an action, provide values for each of its required parameters and, if desired, confirm action before it is executed. A good UI will also allow users to cancel the action. These are interaction tasks that any UI must support. Figure 4 shows how these interaction tasks relate to application concepts: the select, cancel, and confirm tasks are directly associated with the application action, while select-object and select-position interaction tasks are connected with the parameters for which they provide values.

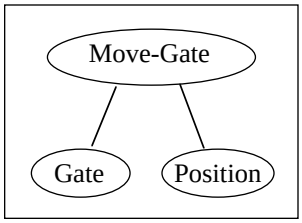

Figure 3. The move-gate action

Each interaction task can be completed in a variety of ways, by using different interaction techniques. Figure 4 illustrates a design in which the move-gate action is selected from a menu, the gate and the position are selected by pointing (mouse-clicking over an object or a position), and the confirmation is also done by using a menu. The action can also be cancelled by using a menu.

The design shown in Figure 4 is only one of many different UIs that can support the move-gate action. Figure 5 shows another design corresponding to a different look-and-feel, a direct manipulation interaction style. In this design, the action is not selected from a menu, but is implicit in the object selection. Thus, when a mouse button is pressed while pointing at the object, not only the object is selected but the action is activated too. To accomplish this, the mouse-press interaction technique is linked to another object, which serves as an adapter connecting it to two different interaction tasks. A gate is selected by the mouse-press and dragged (by moving the mouse) to its new position until the mouse button is released, when the action is completed (a button release confirms the action). It is not possible to cancel the action in this design. Interaction tasks remained the same, as can be seen by comparing figures 4 and 5, but they are connected to different interaction techniques, resulting in a different look-and-feel.

This example illustrates the basic concepts of the TACTICS model: composition and transformations. A UI is composed from a predefined set of components, and this can be done automatically based on the knowledge about application semantics and the UI domain. Different UIs are produced by transforming underlying UI structures –

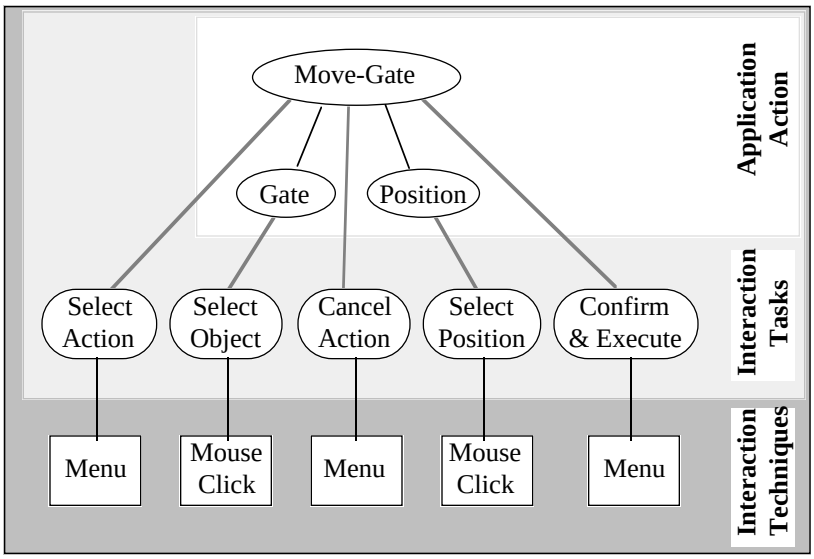

Figure 4. The move-gate action with UI components.

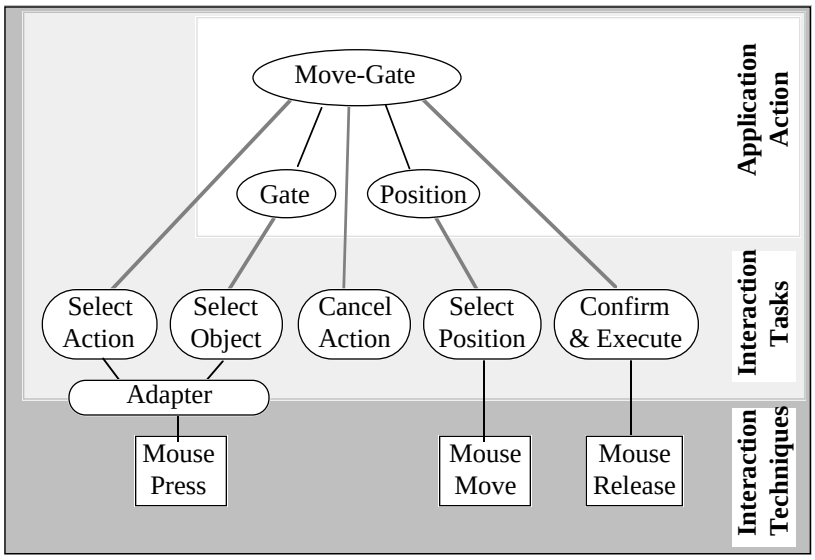

Figure 5. The move-gate action with a different UI

changing UI components and links between them. The representation shown in Figures 4 and 5 is a simplified view of the TACTICS model. The content of the innermost (white) rectangle in the figures corresponds to the high-level description of the application semantics. The rest of the structure, UI components, is automatically generated by the TACTICS composition rules. A transition from Figure 4 to Figure 5 – replacing one set of UI components with another and restructuring them – is done by transformations.

5. USER INTERFACE FUNCTIONALITY

Figures 4 and 5 provide a simplified view of an application and its UI. They consider only inputs and how those inputs are provided. For instance, nothing is said about syntax – when the move-gate action is available and when it is not, and in which order its parameters can be provided. Similarly, providing feedback and objects presentations are not mentioned, nor is maintaining the application and UI context. The example is thus closer to the view of UIs as transducers of inputs to applications that, behind the scene, do the rest of the work. The TACTICS model actually goes much further, moving the boundary between the application-specific and UI-related functionality.

Moving the boundary increases reusability and flexibility of a UI; we can package more support in a UI tool and change a UI in more ways without requiring changes in the application specific part. The boundary is moved by extracting functions which are not application-specific out of an application and into a UI. This does not mean that those functions are application-independent, but only that they commonly occur in applications and that any support for those functions provided by a UI tool may benefit a large number of applications.

These functions are generalized to become reusable building blocks of an application, or more specifically, of its UI. The functioning of each block does not have to be application-independent; application specificity is achieved by setting the properties of individual blocks and thus tuning their performance to conform to the application requirements.

Figure 6 illustrates this. For an end-user, everything inside the dotted rectangle is an application. A UI is the part of an application in charge of communication with the user. The TACTICS model moves the boundary between a UI and the application-specific part by identifying a layer of components that maintain and utilize the UI-related context of an application. In that sense it goes beyond the traditional UI tools and models that focused only on input and output communication tasks. We identified a set of buffering components that keep the UI context and a set of supporting control components that maintain and utilize the contextual information. Consequently, in TACTICS, a UI is no longer only a passive transducer between the user and the application's functional part but it performs three major tasks: communication, buffering, and controlling and maintaining information flow.

There is a functional component for each of the three tasks, and a UI as a whole functions through their interaction. These three types of components are necessary parts of a UI, because they perform mandatory UI tasks. A UI can also include other components, for performing optional tasks, such as error-recovery and help. These tasks belong to the UI because they do not contribute to the application's functionality, but to the quality of its UI. On the other hand, they are optional since a UI can function without them, though a user will probably perceive such a UI of lesser quality.

The *communication* component is in charge of accepting inputs from a user, and presenting information back to the user; it covers what was traditionally considered a role of a UI. The communication component does not pass information directly to the functional part. For instance, when a user enters a parameter value, the value is not necessarily passed to the functional part right away, but it may be stored in the UI until all values are collected. This way, the application's functional part does not depend on the order in which values are entered – it is shielded from a specific syntax which can be changed without having to change the functional part. The values are kept in the *buffering* component, which maintains the current context of each action and the state of a UI as a whole. The *control* component manages the information traffic through the UI; it maintains the information flow between components, controlling who gets which information, and where it comes from.

The flow of information between the UI and the application functional part, and between the UI and the user does not have to be the same. A UI can decouple the order in which information is entered and the order in which it is passed to the functional part. In addition, default and global values break into direct correspondence between the information content that a user specifies and the information content the UI passes to the application functional part. If a

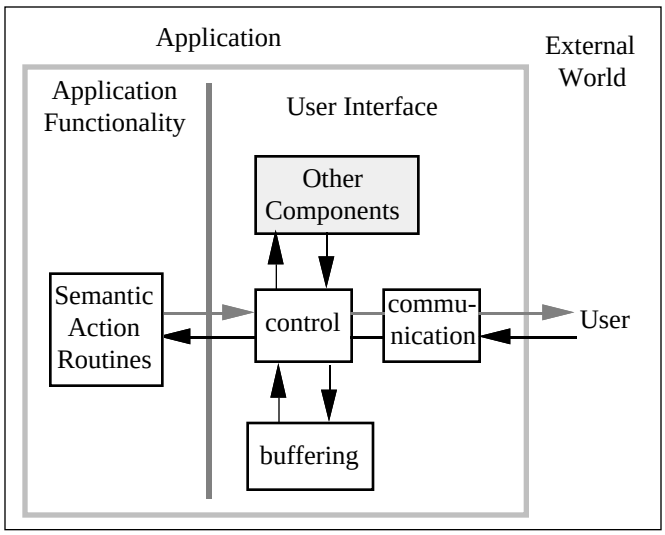

Figure 6. Moving the boundary between the application and the UI functionality

parameter has a default value, it is optional and the user does not have to specify it; if the user chooses not to, the UI will use a designer defined default value when passing information to the functional part. To the functional part, it is transparent whether a value comes from the user, or the UI has obtained it in some other way. In the case of global values, the UI designer does not specify the value at design time, rather the user does this at run time, using special actions provided by the UI designer. With default values, we have a situation when a UI passes the information to the functional part even when the user does not specify it. In the case of global values, we have the opposite situation as well – the UI does not pass to the application the information the user did specify, but stores it internally. This is an important property of a UI, that it can store information and use it at some later stage, or even reuse it repeatedly.

In the same way a UI can buffer and reorder user inputs, it can buffer outputs when providing feedback. A situation when this may be useful is when an object goes through a sequence of changes, and interim states are not of interest, hence require no in-between feedback. A contrasting example is the animation of a transition from one state to another when UI interpolates the interim states an object has to go through, thus generating more feedback than asked for by the functional part.

A UI can control timing of feedback events with respect to the user inputs causing them. Feedback can be provided immediately after each user input, or be delayed until after a sequence of user inputs is completed. Thus, when moving an object, feedback can be given for each new position, or only after the final position is selected. More specifically, in the Circuit Design application, when moving a gate, the gate can follow the cursor, or simply jump to a new position. Again, these variations are the result of different UIs; the functional part remains the same, but the UI can buffer and reorder input and output events.

6. UI COMPONENTS

The three components of a UI (Fig. 6) perform variety of tasks and are further subclassed according to their specific role and properties. Each primitive is defined by a set of properties which control its behavior, methods which deliver that behavior, and messages it can receive from or send to other primitives. Properties are set when configuring an object to serve a specific role, depending on application semantics and the desired dialogue style. Properties can be modified by design transformations. Figure 7 shows a partial class hierarchy of UI modeling primitives.

Communication components. They are subclassed into interaction techniques and presentation objects. *Interaction techniques* map external input into internal form. An interaction technique can be simple, such as "push button", or composite, such as "drag: push button, move mouse, release the button". *Presentation objects* map a given internal information form into desired presentation properties, such as color, size, shape, or tone pitch.

Buffering components. This category has three subclasses, each maintaining a different aspect of the overall UI context: action-context, object-context, and global-context.

During the activation of an action, the *action-context* primitive keeps all (known) pieces of information needed to perform the action, decoupling the order in which information is specified from the order in which it is passed to the application functional part. Between the activations, the action-context instance keeps the information which is reusable across multiple activations of the action and specific to that action, such as local default values and partially-global values for action parameters. It also serves as a local blackboard where control primitives pertaining to the same action keep information needed for their synchronization, e.g. to enforce specific sequencing constraints.

The *object-context* primitive keeps global properties for each application object class. These include default attribute values and global attribute values.

The action and the object related buffering primitives implicitly describe some aspects of the UI state; the context not captured there is maintained in the *global-context* primitives. They serve as a public blackboard and information posted there is accessible to all, which is not the case with the action-context and object-context primitives. The postings include the state information for maintaining the sequencing and information flow control, and properties of sets, e.g. currently selected objects, and clipboard objects.

Control components. Their role spans three major tasks and there is a subclass corresponding to each of the three tasks: information flow control, sequencing control, and event propagation subclass.

The first subclass, *information-flow-control* (IFC), maintains information flow among primitives, integrating functional parts of a UI into a single structure. Its primitives act as intermediaries in obtaining information required by the action. They know where to look for each piece of infor-

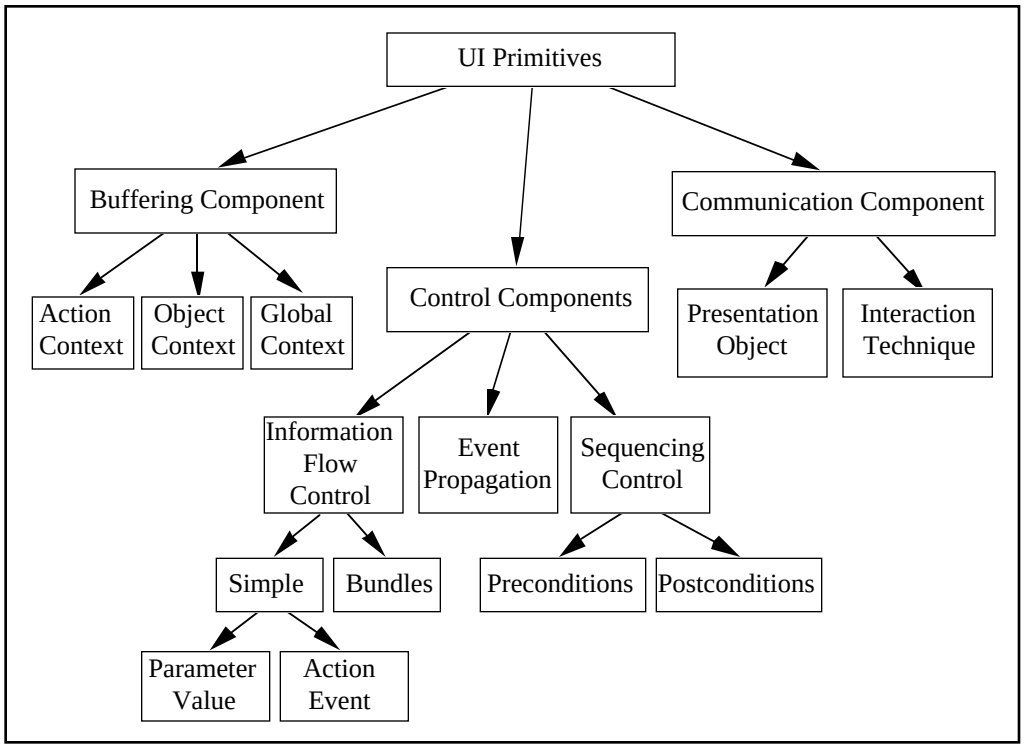

Figure 7. Partial class hierarchy of UI modeling primitives

mation, whether to get it from a communication primitive, or from a buffering component, and which one. They are specialized according to the nature of the information they pass and what the information is used for. The interaction task components in figures 4 and 5 belong to this category. More specifically, select-, cancel- and confirm-action are action-event primitives, and select-object and select-position are parameter-value primitives. There are no action-event primitives for enabling/disabling an action because the user does not enable/disable actions directly, but it is a side effect of other actions.

Whether the user has to explicitly signal events for a primitive depends on how the primitive is configured. For instance, the confirm-action event is specified only if confirmation is required, otherwise the primitive will "fire" automatically as soon as all necessary information is collected and will pass the information to an appropriate semantic action routine.

Another subclass of IFC components, *bundles*, handles user inputs that result in several parameter values or action events, or can be interpreted in more than one way. For instance, the adapter in Fig. 5 connecting the mouse-press interaction technique with the select-action and select-object interaction tasks is actually an instance of bundle.

The *sequencing* control primitives maintain and monitor the relevant UI context. They update the context whenever something potentially affecting IFC primitives happens, and they constantly evaluate the context to enable/disable those primitives. The two tasks are performed by specialized primitives: *postconditions* are updating the context, while *preconditions* are evaluating it.

Finally, the *event-propagation* primitives propagate events of interest, possibly performing relation detection and enforcement. While pre- and post-conditions also propagate events when enabling or disabling other primitives, they do it indirectly, through context updates. The event-propagation primitives do this directly – they monitor an event of interest and, when it happens, propagate it right to a desired target. By doing so, they effectively link other primitives and establish a flow of information between them. In that sense, they complement IFC primitives – they are specialized to monitor specific events, which do not have to originate from the user inputs.

6.1. Example Revisited

Figures 4 and 5 presented a simplified view of UI structures, with only a subset of primitives needed to implement a UI. Figure 8 expands the UI part of Figure 4 by including other primitives introduced in the previous section: pre- and post-conditions and buffering components. Also shown are the event-propagation and presentation-object primitives, and the semantic action routine that implements functionality of the move-gate action. The global-context, object-context, presentation-object, and event-propagation primitives are not instantiated exclusively for the move-gate action, but are shared with other actions.

The structure shown in Fig. 8 is automatically generated by the TACTICS composition rules. Whereas the rules generate a unique name for each component, here generic names are used to make comparing figures 4 and 8 easier.

Figure 8 also includes an extra interaction technique, type-in, to illustrate that multiple techniques can serve an IFC primitive. For instance, this enables users to select the action not only from a menu, but also typing its name or an accelerator.

Without going into all details of each individual primitive, we point out how they interact and how the structure as a whole implements a specific UI for the move-gate action.

Arrows in Fig. 8 indicate the flow of information. For instance, interaction techniques pass values to IFC primitives, which pass them on to the action-context primitive (where it is stored for later use), and/or to their postconditions

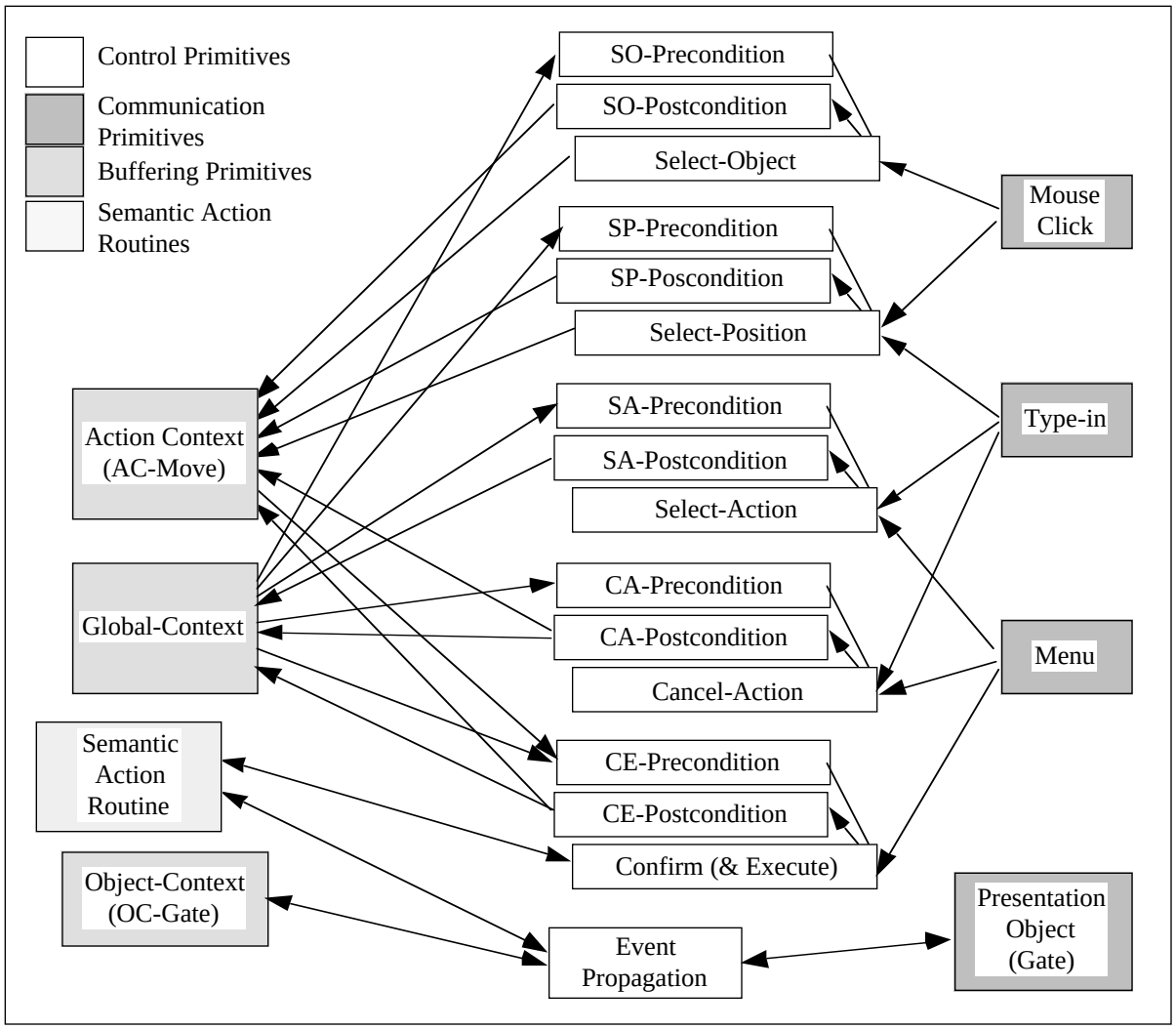

Figure 8. A UI structure for the *move-gate* command

which in turn may pass (possibly processed) values on to one or more of buffering primitives. IFC primitives can accept input from interaction techniques only when their preconditions are satisfied. Whether a postcondition passes a value to the action-context or to the global-context primitive depends on whether it is meant only for preconditions belonging to the same action, or to other actions as well. Preconditions retrieve values from buffering primitives. IFC primitives and their preconditions are connected by plain lines because there is no information exchange between them.

Pre- and postconditions can be configured to implement different syntaxes. For a prefix syntax, select-action has a postcondition asserting that the action is selected, and all other primitives have corresponding preconditions. Select-action has preconditions derived from semantics preconditions of the move-gate action as a whole. Each parameter-value primitive has a postcondition asserting that its parameter has a value, and confirm&execute has corresponding preconditions. As a consequence, the action cannot be executed before it is selected and all its parameters have values. Transformations can change syntax by manipulating pre- and postconditions. What is important is that the semantic routines do not depend on a specific syntax or other UI details and can be reused in different UI designs.

7. IMPLEMENTATION

A prototype of the TACTICS tool has been implemented and tried on two example applications, Circuit Design and File Browser. The tool generates the structure shown in Fig. 8 automatically – the tool instantiates and configures all primitives driven by the description of application semantics. Changing a UI is done by applying transformations. For instance, the design in Fig. 4 is changed to one in Fig. 5 by applying a bundling transformation and changing input bindings. Another transformation, not visible in the structures shown but needed to configure the select-position primitive to initiate feedback whenever there is a new value, was to change feedback style of the position parameter to dynamic. The tool also contains a set of consistency rules which, for instance, detect if the same interaction ("click on the object") is used to activate two different actions (e.g. move and rotate). Implementation is in Inference Corp's Art4 and Common Lisp Object System

(CLOS), running on SparcII under X windows.

ART was chosen as the implementation platform because it allows combining rule-based and object-oriented programming. All primitives are defined as ART schemata, which makes subclassing easy, and also makes the resulting structure visible to pattern-matching rules that check designs for conflicts and inconsistencies. Behavior of a primitive is implemented by methods attached to its corresponding schemata and by active values and pattern-matching rules. The *event-propagation* primitives are completely implemented via active values and pattern-matching rules.

8. CONCLUSIONS

A model-based approach to UI design addresses the major requirements on the UI tools: improved models not only expand the range of interfaces a UI design tool can produce, but also capture knowledge the tool needs to assist in a design process – creating initial designs, changing designs, and providing guidance along the process.

The paper presented the TACTICS model of human-computer interaction, which integrates the compositional and transformational approach to generating and exploring UIs. TACTICS supports the automatic generation of UIs without sacrificing the range of possible designs. At the same time, it reduces dependence between the UI and the application, allowing the UI part to be changed in more ways without affecting the application functionality. This was made possible by shifting the boundary between the application-specific and UI-related functionality. The TACTICS model moves this boundary by identifying a layer managing the UI-related context of the application – buffering components that keep the UI context and a set of supporting control components that maintain and utilize the contextual information. This results in increased reusability and flexibility of UIs – more support can be packaged into a UI tool, with more ways to change a UI without requiring changes in the application part.

Acknowledgements

The Graphics and User Interface Research Group at the George Washington University provided an intellectual environment in which this work originated. I am grateful to James Foley and other members of the UIDE research team for their discussions concerning some of these ideas. Financial support was provided by National Science Foundation grant IRI-8813179, Inference Corporation, Software Productivity Consortium, Siemens Corporation, the Department of EE & CS Industrial Liaison Program, and the Graphics, Visualization, and Usability Center at the Georgia Institute of Technology.

REFERENCES

[Balzer 85] Balzer, Robert, "A 15 Year Perspective on Automatic Programming," *IEEE TSE*, Vol. SE-11, No.11, November 1985, pp. 1257-1268.

[Bleser 90] Bleser, T., and Sibert J. Toto: A Tool for Selecting Interaction Techniques. In Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology. Snowbird, Utah, 1990, pp. 135-142.

[Card 90] Card, S.K., Mackinlay J.D., and Robertson G.G. The Design Space of Input Devices. In Proceedings of CHI'90. ACM New York, 1990, pp. 117-124.

[Darlington 81] Darlington John, "An Experimental Program Transformation and Synthesis System," *Artificial Intelligence* **16**, 1986, pp. 1-46.

[deBaar 92] deBaar, D.J.M.J., Foley, J.D., and Mullet, K.E., "Coupling Application Design and User Interface Design," in *Proceedings of CHI'92 (Monterey, CA, May 3-4, 1992)*, ACM New York, 1992.

[Foley 87] Foley, J., C. Gibbs, and W. Kim, "Algorithms to Transform the Formal Specification of a User-Computer Interface," in *Proceedings INTERACT '87, 2nd IFIP Conference on Human-Computer Interaction*, Elsevier Science Publishers, Amsterdam, 1987, pp. 1001-1006.

[Foley 91] Foley, J., Kim W., Kovacevic S., and Murray K. UIDE – An Intelligent User Interface Design Environment. In Sullivan, J. and Tyler, S. (eds.), Architectures for Intelligent Interfaces: Elements and Prototypes, Addison-Wesley, 1991.

[Foley 91a] Foley, J., Gieskens, D., Kim W.C., Kovacevic S., Moran, L., and Sukaviriya, P. A Second-Generation Knowledge Base for the User Interface Design Environment. Technical Report GWU-IIST-91-13, Dept. of EE&CS, The George Washington University, Washington, D.C. 20052, 1991.

[Green 85] Green, M. The University of Alberta UIMS. In Proceedings SIGGRAPH '85, published as Computer Graphics, 19(3), 1985, pp. 205-213.

[Hayes 85] Hayes, P., Szekely P., and Lerner R. Design Alternatives for User Interface Management Systems Based on Experience with COUSIN. In Proceedings of CHI'85, ACM, New York, 1985, pp. 169-175.

[Kim 90] Kim, W.C., and Foley, J., "DON: User Interface Presentation Design Assistant," in *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, Oct. 1990.

[Kovacevic 92] Kovacevic, S. A Compositional Model of Human-Computer Dialogues. In Blattner M. and Dannenberg R. (Eds), Multimedia Interface Design, ACM Press, 1992.

[Kovacevic 92a] Kovacevic, S. A Compositional Model of Human-Computer Interaction. DSc dissertation, Dept. of EE&CS, The George Washington University, 1992.

[Myers 90] Myers, B.A., A New Model for Handling Input, ACM TOIS 8(3), July 1990, pp.289-320.

[Neches 93] Neches, R., Foley, J., Szekely, P., Sukaviriya, P., Luo, P., Kovacevic, S., and Hudson, P. Knowledgeable Development Environments Using Shared Design Models. In Proceedings of 1993 International Workshop on Intelligent User Interfaces, ACM New York. Orlando, FL. 1993. pp.63-71.

[Olsen 86] Olsen, D. MIKE: The Menu Interaction Kontrol Environment. ACM TOG 5(4), Oct. 1986, pp.318-344.

[Olsen 89] Olsen, D., "A Programing Language Basis for User Interface Management," in *Proceedings of CHI'89 (Austin, Texas, April 30-May 4, 1989)*, ACM New York, 1989, pp.171-176.

[Partsch 83] Partsch, H., and R. Steinbrüggen, "Program Transformation Systems," ACM Computing Surveys, 15(3), September 1983, pp.199-236.

[Puerta 92] Puerta, A., Eriksson, H., Egar, J., and Musen, M., *Generation of Knowledge-Acquisition Tools from Reusable Domain Ontologies*, Report KSL-92-81, Knowledge Systems Laboratory, Stanford University, Stanford, CA 94305-5479. 1992.

[Singh 89] Singh, and Green M. A High-Level User Interface Management System. In Proceedings of CHI'89. ACM New York, 1989, pp. 133-138.

[Sukaviriya 90] Sukaviriya, P., and Foley J. Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help. In Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology. Snowbird, Utah, 1990, pp. 152-166.

[Szekely 91] Szekely, P. Using Classification and Separation to Build Intelligent Interfaces. In Sullivan, J. and Tyler, S. (eds.), Architectures for Intelligent Interfaces: Elements and Prototypes. Addison-Wesley, 1991.

[Szekely 92] Szekely, P., Luo, P., and Neches, R., Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design, in Proceedings of CHI'92. ACM New York, 1992.

[Wiecha 90] Wiecha, C., Bennet, W., Boies, S., Gould, J., and Greene S., ITS: A Tool for Rapidly Developing Interactive Applications, ACM TOIS, 8(3), July 1990. pp.204-236.